Chapter 1
The six essentials of software testing

This chapter describes the fundamentals for engineering a new software testing process or renovating one currently in existence. The software testing process is the means by which people, methods, measurements, tools, and equipment are integrated to test a software product. The essentials of the software testing process that serve as the foundation for this book are:

(1) The quality of the test process determines the success of the test effort.
(2) Prevent defect migration by using early life-cycle testing techniques.
(3) The time for software testing tools is now.
(4) A real person must take responsibility for improving the testing process.
(5) Testing is a professional discipline requiring trained, skilled people.
(6) Cultivate a positive team attitude of creative destruction.

Essential 1: The quality of the test process determines the success of the test effort

The quality of a software system is primarily determined by the quality of the software process that produced it. Likewise, the quality and effectiveness of software testing are primarily determined by the quality of the test processes used.

Testing has its own cycle. The testing process begins with the product requirements phase and from there parallels the entire development process. In other words, for each phase of the development process there is an important testing activity.

Test groups that operate within organizations having an immature development process will feel more pain than those that don't. But regardless of the state of maturity of the development organization, the test group can and should focus on improving its own internal process. An immature test

process within an immature development organization will result in an unproductive, chaotic, frustrating environment that produces low-quality results and unsatisfactory products. People effectively renovating a testing process within that same immature organization will serve as a catalyst for improving the development process as a whole.

Essential 2: Prevent defect migration by using early life-cycle testing techniques

More than half the errors are usually introduced in the requirements phase. The cost of errors is minimized if they are detected in the same phase as they are introduced, and an effective test program prevents the migration of errors from any development phase to any subsequent phases.

While many of us are aware of this, in practice we often do not have mechanisms in place to detect these errors until much later – often not until function and system test, at which point we have entered "The Chaos Zone". Chances are that we are currently missing the best opportunity for improving the effectiveness of our testing if we are not taking advantage of the proven testing techniques that can be applied early in the development process.

For example, we should learn to perform reviews on critical documents like requirements. Even an immature organization can implement an effective inspections program. Inspections have proven a winner repeatedly for well over 20 years, and have been justified (and documented) in cost/benefit terms over and over again.

Essential 3: The time for software testing tools is now

After many years of observation, evaluation, and mostly waiting, we can now say the time for testing tools has arrived. There is a wide variety of tool vendors to choose from, many of which have mature, healthy products.

For example, test-cycle time reductions and automation providing for 24 hours a day of unattended test operations can be achieved with capture/playback tools. There is one for every major platform: some are more intrusive than others; some are appropriate for client/server; some are technically elaborate to use; others are simple.

Another type of essential tool is the structural coverage tool. Used to determine if the software has been thoroughly tested, this tool tells us specifically which parts of the product have in fact been executed by our tests. It is

no longer acceptable to expect our customers to be the first to execute our code!

It is important to have a strategy for tool acquisition, and a proper procedure for handling tool selection. While such procedures are based on common sense, they do need to be systematically implemented. Tool acquisition is an area where there may be a strong case for seeking independent expert advice.

Essential 4: A real person must take responsibility for improving the testing process

If the testing group is feeling pain, start campaigning for improvements to a few of the key issues, such as better specifications, and better reviews and inspections.

Management should appoint an architect or small core team to prioritize potential improvements and lead the testing improvement effort, and must make it clear that they will give their ongoing support. It is not rocket science, but it takes effort – and time. Tools can help tremendously, but they must be used within an overall test process that includes effective test planning and design.

When all is said and done, software testing is a process that requires people who take responsibility for its improvement. For the testing process to improve, someone must plan and manage the progress.

Essential 5: Testing is a professional discipline requiring trained, skilled people

Software testing has become a profession – a career choice – and a place to make our mark. The software testing process has evolved considerably, and has reached the point where it is a discipline requiring trained professionals. To succeed today, an organization must be adequately staffed with skilled software testing professionals who get proper support from management.

Testing is not an entry level job or stepping stone to other things. Many people find that when done properly, it surpasses the challenge of product development. It should be organized so it can faithfully maintain allegiance to the customer. It should not be subservient to or be positioned for easy overrule by the product development or any other organization.

Testing should be independent, unbiased, and organized for the fair sharing of recognition and rewards for contributions made to product quality.

Essential 6: Cultivate a positive team attitude of creative destruction

Testing requires disciplined creativity. Good testing, that is devising and executing successful tests – tests that discover the defects in a product – requires real ingenuity, and may be viewed as destructive. Indeed, considerable creativity is needed to destroy something in a controlled and systematic way. Good testers are methodically taking the product apart, finding its weaknesses, pushing it up to, and beyond, its limits.

Can we make this product fail to do what we expect it to do? When is it going to break? We know it's going to break, because we can always find errors. But is it breaking at reasonable boundaries? Is it stressing or breaking acceptably or unacceptably, given its criticality? Do we know that we have covered all the important possibilities? It is the testers who supply this information, and it is only when we're satisfied that these questions, and many others, have been properly addressed, that we should ship the product to our customer.

Establishing the proper "test to break" mental attitude has a profound effect on testing success. If the objective is to show that the product does what it shouldn't do, and doesn't do what it should, we're on the way to testing success. Although far from the norm today, the results that we get when practitioners and their managers together cultivate this attitude of disciplined, creative destruction are nothing less than astonishing.

Successful testing requires a methodical approach. It requires us to focus on the basic critical factors: planning, project and process control, risk management, inspections, measurement, tools, organization – and professionalism. Remember that testers make a vital positive contribution throughout the development process to ensure the quality of the product.

Chapter 3

The clean-sheet approach to getting started

In the following chapters the essentials of testing are developed in more detail as well as many techniques and ideas that are independently adaptable to any organization. Start now with a clean sheet of paper and keep a running list of potential improvements you believe are key to renovating the software testing process in your organization. These ideas for improvement may take many forms: a new idea you want to use; a new technique or tool you want to investigate; a fundamental change in the software test process. Consider objective, practical, cost-effective goals. Review and update this list from time to time as you read this book, and prioritize the entries. To get you started, a few suggestions are listed below. Others are given at the end of several of the chapters which follow.

Potential improvements

- Investigate what it would take to implement an effective inspections program.
- Launch an effort to determine what tools would provide the most leverage.
- Begin today to cultivate a "test to break" attitude of creative destruction.

At the end of the book, Appendix H gives a list of all the "Improvements to be implemented" created by a group of 11 testing professionals. These professionals came from widely different types of organizations with different levels of software engineering process maturity, and the list represents what they considered they could realistically attempt within their own organizations after attending the software testing course on which this book is based.

A second list in Appendix H was compiled during a testing course held on-site at a medium-sized software company. This time the list is organized under two headings: "Prioritized improvements internal to the software testing function" and "Prioritized improvements external to the software testing

Chapter 2

The state of the art and the state of the practice

For centuries our effort to understand and control our environment by complex disciplines has evolved largely through painful trial and disastrous error. Eventually, basic know-how, guidelines and best practices evolve, and become second nature to practicing professionals in their particular fields.

In the evolutionary scale of human disciplines, software engineering still flounders in the primeval mud. When it feels like this in our own organization, it's worth remembering how young the software development discipline really is. Even the phrase "software engineering," implying a systematic approach with ground rules, was not used until the late 1960s.

Furthermore, the gap between the state of the art and the state of the practice is a chasm that, if anything, becomes wider. Software testing, for instance, has yet to become a fundamental component of university software engineering curricula, and training in industry is improving but often haphazard. Papers are published (though not always read by those who need them most) presenting leading-edge ways of doing software better and achieving desirable levels of software quality in particular, but many of these state-of-the-art methods are unproven in the field, and many omit real-world dimensions like return on investment.

Even the many well-proven methods are largely unused in industry today, and the development of software systems remains inordinately expensive, fraught with costly errors, and often late, with products often costing far more to maintain over their lifetime than to develop in the first place.

At the same time, the systems we build are ever more complex and critical, and more than 50% of the development effort is frequently spent on testing. Yet many of the people responsible for this testing are still struggling for a viable working platform, properly equipped with useful, available test tools and an independent, properly organized and resourced position in the software development environment.

The short eventful history of a very new discipline

The notion of software engineering implies, among other things, an attempt to achieve reliability, functionality, predictability, economy, and efficiency. Software testing concerns itself with all of these.

In the early days of software development, this concern was rather narrowly interpreted. Testing was regarded as "debugging", or fixing the known bugs in the software, and was usually performed by the developers themselves. There were rarely any dedicated resources for testing, and those that were dedicated got involved very late in the development process, often only after the product was coded and nearly complete. In the worst examples, this is still the case.

By 1957 software testing was distinguished from debugging and became regarded as detecting the bugs in the software. But testing was still an after-development activity, with the attitude of "let's push it around a little bit just to show ourselves that the product actually works." The underlying objective was to convince ourselves that it worked – and then ship it. The universities did not talk much about testing either. Computer science curricula dealt with numerical methods and algorithm development, but not with software engineering or testing. First compilers, then operating systems and then data bases were the prime focus, but none of these were helpful in getting people to address testing issues in the real world of industry.

By the 1970s "software engineering" as a term was used more often, though there was little consensus as to what it really meant. The first formal conference on testing was held at the University of North Carolina in 1972, and a series of publications followed (Hetzel, 1973; Myers 1976, 1979).

It was Myers (1979) who saw that the self-fulfilling nature of human goals had a major impact on testing work. He defined testing as "the process of executing a program with the intent of finding errors." He pointed out that if our goal is to show the absence of errors, we will discover very few of them. If our goal is to show the presence of errors, we will discover a large number of them. Establishing the proper goal and mind-set has a profound effect on testing success (see Chapter 4).

The work of Myers and others in the 1970s was a major step in the development of satisfactory testing processes, but in the real world of industry, testing continued to be at the top of the list of things that got dumped when the schedules and the budget became tight. Testing was started too late in the process, and there was not enough time to do it properly when it did

start. Management would even short-circuit it altogether with, "well let's just ship the product anyway because we don't have time to test it" (i.e., because the rewards appear to outweigh the risks). If this sounds at all familiar, it is because in many organizations this is still common practice.

By the early 1980s "Quality!" became the battle cry. Software developers and testers started to get together to talk about software engineering and testing. Groups were formed that eventually created many of the standards we have today. These standards, like the IEEE (Institute of Electrical and Electronics Engineers) and ANSI (American National Standards Institute) standards in the USA or the ISO (International Standards Organization) international standards, are now becoming too weighty to digest in their full published form for everyday practical purposes. However, they do include important guidelines, a good baseline for contracts, and provide an invaluable reference.

In the 1990s testing tools finally came into their own. It is now widely understood that tools are not only merely useful, but absolutely vital to adequate testing of the systems built today, and we are seeing the development of a wide variety of tools to help with the testing process. By now almost every company developing software has somebody looking into tools, and tools have become a critical part of the testing process. However, on their own, like most techniques, they don't solve the real problem.

Development and testing evolution

	1960	1970	1995
Software size	small	moderate	very large
Degree of software complexity	low	medium	high
Size of development teams	small	medium	large
Development methods and standards	*ad hoc*	moderate	sophisticated
Test methods and standards	*ad hoc*	primitive	emerging
Independent test organizations	few	some	many
Recognition of testing's importance	little	some	significant
Number of testing professionals	few	few	many

Despite the enormous advances in the last 30 years, the software process (including the testing process) in most companies is still very immature. Furthermore, the complexity and criticality of the problems which software is expected to solve have become greater, and platform complexities have become bigger, and this is tending to out-run our ability to put more effective methods, tools, and professionals in place. It is no wonder that life gets increasingly tough for testers.

Where exactly are we now?

A proper understanding of the concept of the maturity of our software engineering practices is fundamental to our testing success. It is essential to know our present baseline and we need an objective assessment of the maturity level of the development environment in which we are operating and its impact on our test process.

During the 1980s, the Software Engineering Institute (SEI) at Carnegie Mellon University developed a software process assessment method and capability maturity model for the United States Department of Defense. Their 1987 technical report proposed a five-level capability maturity model. This is a practical model for organizations to measure their own level of software maturity.

The reality is that most organizations still have fairly immature software process development environments. About 75% of the world of software development is at level 1 on a scale of 1–5 in terms of really having mature software processes in place. When we talk about software testing in the real world, this 1 on the scale of 1–5 represents the actual environment in which we work and within which we have to do our testing.

The good news is that the SEI model also explains how to get from each level to the next. There are companies that have achieved level 5 status but they are in a very small minority (see Chapter 6 for more detail on the SEI and software engineering maturity, and Chapter 14 for current trends and best practice literature).

How should testing be positioned?

When testing was something that was done after development, a back-end activity after the code was finished, the development life cycle consisted of requirements, design, implementation and then test. We waited until the code was done and then we did some function testing. Errors were easy to find, and, not surprisingly, developers and testers often came to be seen as adversarial.

Once it was understood that testing was much more than debugging, it was also clear that testing was more than just a phase near the end of the development cycle. In fact early testing is a key to the success of the test effort. Testing has a life cycle of its own, and there is useful and constructive testing to be done throughout the entire life cycle of development (see Chapter 5).

We are talking about testing as the vehicle for detecting errors in the software, but we will use the term software to mean much more than just the code. The definition of software should include the related documentation such as the specifications, the design documents, and the user manuals.

The notion of engineering software quality means not just reliability and efficiency in the abstract, but something more closely linked to customer requirements and customer satisfaction. Furthermore, how does testing as carried out in most organizations today relate to quality assurance and the quality assurance (QA) function?

QA is not the same as process development. The QA function does not generally include moving the process ball forward, i.e., being a process expert and suggesting ways of implementing process improvements. QA is usually defined in the literature as a function which:

- monitors the software and the development processes that produce it;
- ensures full compliance with established standards and procedures for the software and the software process;
- ensures that inadequacies in the product, the process, or the standards are brought to management's attention.

In contrast to QA, testing performs in-depth analysis, testing, and overall evaluation of a software product or system. It is meant to represent the customer and as such primarily owes its allegiance to the customer rather than to development. This is widely understood, and yet in many organizations testing is still formally subordinate to development.

This subordination to development means the voice of testing often goes unheard. While the testing people do their job and report the results, they frequently find their management simply thanks them for their information, but ships the product anyway. Little or no real action is taken based on the testers' input, because testing often has no authority or power or control in the organization. This is frequently reflected in staffing policies, where it is sometimes assumed that testing is a place to put junior people or anyone who can be spared without their loss being felt elsewhere.

However, things are improving. The world understands, at least in theory, the importance of testing, the importance of having an independent testing function with a viable position in the organization, and the necessity to hire technically competent people and reward them adequately (see Chapter 13). Testing is coming to be seen as a profession with its own work products that are maintained as important assets of the organization (see Chapter 6).

References

Hetzel, W. (1973). *Program Test Methods*. Englewood Cliffs, NJ: Prentice-Hall.
Myers, G.J. (1976). *Software Reliability: Principles and Practices*. John Wiley.
Myers, G.J. (1979). *The Art of Software Testing*. John Wiley.

Chapter 3

The clean-sheet approach to getting started

In the following chapters the essentials of testing are developed in more detail as well as many techniques and ideas that are independently adaptable to any organization. Start now with a clean sheet of paper and keep a running list of potential improvements you believe are key to renovating the software testing process in your organization. These ideas for improvement may take many forms: a new idea you want to use; a new technique or tool you want to investigate; a fundamental change in the software test process. Consider objective, practical, cost-effective goals. Review and update this list from time to time as you read this book, and prioritize the entries. To get you started, a few suggestions are listed below. Others are given at the end of several of the chapters which follow.

Potential improvements

- Investigate what it would take to implement an effective inspections program.
- Launch an effort to determine what tools would provide the most leverage.
- Begin today to cultivate a "test to break" attitude of creative destruction.

At the end of the book, Appendix H gives a list of all the "Improvements to be implemented" created by a group of 11 testing professionals. These professionals came from widely different types of organizations with different levels of software engineering process maturity, and the list represents what they considered they could realistically attempt within their own organizations after attending the software testing course on which this book is based.

A second list in Appendix H was compiled during a testing course held on-site at a medium-sized software company. This time the list is organized under two headings: "Prioritized improvements internal to the software testing function" and "Prioritized improvements external to the software testing

function." It is a good idea to identify the improvement ideas on your sheet as being internal or external to the testing group. It can also be useful to rate items to implement in terms of:

- the difficulty of implementing in the organization;
- the resources required for implementation;
- the payoff in improvement to the testing process;
- the short-, medium- or long-term effort required to (realistically) achieve the item.

This rating can be done as a second pass after the potential ideas have been assembled.

Remember to include long-term goals, too. Real progress is made in small steps over two, five, 10 years or more. We have discovered a few companies that do long-term plans, and are currently implementing their third 10-year plan!

Keep asking the right question: "Is what we are doing now an improvement over what we did before?" If the answer is yes, and yes again next time and the time after, then we are on the road to continuous improvement. It is surprising how much can be achieved with limited resources. Management and peer support will grow once improvements are visible to the organization.

PART II
The framework for test process improvement

"The will to win is important, but the will to prepare is crucial"

JOE PATERNO, Head Coach, Pennsylvania State University

Chapter 4
Establishing a practical perspective

Chapter 5
Critical choices: what, when, and how to test

Chapter 6
Critical disciplines: framework for testing

Chapter 4
Establishing a practical perspective

Software is written by people, and people make mistakes. In standard commercial software, errors are present. These errors are expensive; some more than others. We can't entirely prevent them being introduced, but we can work to locate them early, especially the most critical ones.

We need to understand and be able to implement appropriate testing techniques. But it is also important to consider some of the non-technical issues that help us toward a more successful testing effort, so we can not only test well, but also test smarter.

Definitions of the testing process and the elements within it are important. They are essential for communication and agreement with our collaborators on what we are doing, why, and how. They establish a common language, providing a foundation for future discussions. They help us get the right focus for our testing effort, so that we get beyond just putting out forest fires.

As testers, what we see as our mission, our basic professional attitude, can have a profound effect on the success of the testing effort and can provide a catalyst for improvement throughout the development process.

What are we aiming for?

What is the ultimate goal of testing? At the end of the line is the fact that we are in the business of making sure our customers are successful. It is our customers who pay the bills, and if we are to stay in business we have to solve their problems. Our underlying goal must be their delight and satisfaction. We aim for quality, but quality isn't just an abstract ideal. We are developing systems to be used, and used successfully, not to be admired on the shelf. If quality is to be a meaningful and useful goal in the real world, it must include the customer.

Testers are not in charge of the whole quality program. However, many testing decisions should ultimately be based on customer satisfaction. In

large and complex development organizations, and in the hurly-burly of most software development projects, this is very easily forgotten. What do our customers want from the system? Will it deliver what they want when they use it? These are the practical questions that lie behind many testing decisions.

All you ever wanted to know about errors

Faults, failures, errors, defects, issues, bugs, mistakes. If these are the targets that testers spend their professional careers hunting down, it's important to know something about them.

The what and the why

Errors lurk in the work products of the software development process. Work products don't just mean code. They include product requirements, functional specifications, drafts and final versions of user manuals, drafts and final versions of data sheets, technical support notices and many other things which are all work products of different phases of the development process.

Software production can be seen as a series of imperfect translation processes. Each of these translations produces a work product or deliverable. Software errors are introduced when there is a failure to completely and accurately translate one representation to another, or to fully match the solution to the problem.

Software errors are human errors. Since all human activity, especially complex activity, involves error, testing accepts this fact and concentrates on detecting errors in the most productive and efficient ways it can devise.

There are several different terms used within this model of general failure to translate successfully (IEEE/ANSI, 1990 [Std 610.12-1990]):

- *Mistake*: A human action that produces an incorrect result.
- *Fault*: An incorrect step, process, or data definition in a computer program. The outgrowth of the mistake. (Potentially leads to a failure.)
- *Failure*: An incorrect result. The result (manifestation) of the fault (e.g., a crash).
- *Error*: The amount by which the result is incorrect.

Mistakes are what people make, and a mistake can be defined as the thing the person did wrong. The phone rang while we were coding, and while distracted we pressed the wrong key and the results of that mistake produced a fault or bug in our work product.

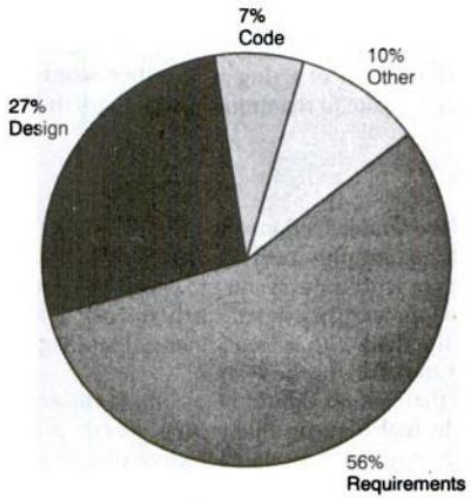

Figure 4.1 Defect distribution. Data obtained from a presentation entitled "Writing testable requirements", by Dick Bender (1993). The word "requirements" in this context includes the functional design. (© 1993, 1994 Software Development Technologies)

Failures are the manifestation of these faults. The fault or bug is there, within the documentation or the code, and even if it hasn't caused a failure yet, it's still a fault and it's the tester's job to find it.

When the failure manifests itself and the system goes down or the user gets the wrong message back or the cash-point provides the wrong amount of money, then according to the above definition, the amount by which that result is incorrect is the error.

Few people use these terms accurately, and in practice this is not very important. For the purposes of this book, defect or error is used most of the time. The important thing is to see that there are different types of things we are looking for when we are testing.

Where are errors?

A closer look at the common distribution of errors will help us to focus our test effort better and in the most productive place. There is strong evidence that errors are concentrated in the earlier stages of the development process, as shown in Figure 4.1.

Errors at the early stage are usually thick on the ground, and they also have the unpleasant habit of migration. Using the analogy of an assembly line in manufacturing, if a bad component or sub-assembly is allowed to

enter the line, the unit to which it is attached is also "bad" from that point on. The problem increases as the unit progresses down the line and out of the door, and so does the cost of fixing it. In other words, errors are not self-contained. As they migrate to the units downstream, they take on new forms.

The cost of errors

All errors are costly. Undetected errors, as well as errors detected late in the development process, are the most expensive of all. Undetected errors will migrate downstream within the system to cause failures. If detected only at a later stage of development they entail costly rework. If not detected, they can cause failures in the field with serious financial and legal consequences, and (at best) high lifetime costs for the system.

This means that testing should be a continuous activity throughout the development cycle. It also means that testing the end product is only one battle – and certainly not the most cost-effective one to win – in the software quality war. Today, anything between 40% and 70% of initial software development time and resources can be devoted to error detection and removal. The bad news is that most organizations have not established a way of knowing what they actually do spend. However, any significant improvement in how testing resources are used can greatly reduce overall development costs. (For further development of the "test early" idea, see Chapter 5.)

So what *is* testing really? Some definitions

Definitions matter, although consensus as to what testing "really is" is less important than being able to use these definitions to focus our attention on the things that should happen when we are testing.

Historical definitions of testing

(1) Establishing confidence that a program does what it is supposed to do (Hetzel, 1973).
(2) The process of executing a program or system with the intent of finding errors (Myers, 1979).
(3) Detecting specification errors and deviations from the specification.
(4) Any activity aimed at evaluating an attribute or capability of a program or system (Hetzel, 1983).
(5) The measurement of software quality (Hetzel, 1983).
(6) The process of evaluating a program or system.
(7) Verifying that a system satisfies its specified requirements or identifying differences between expected and actual results.
(8) Confirming that a program performs its intended functions correctly.

All these definitions are useful, but in different ways. Some definitions (2) focus on what is done while testing (Myers, 1979); others focus on more general objectives like assessing quality (5) and customer satisfaction (1, 3), and others, like (7) and (8), focus on goals like expected results.

For example, if customer satisfaction is our goal (and it should be), this satisfaction, or what would constitute it, should be expressed in the requirements. In situations where requirements are well written and complete, we already have an important advantage. We can ask: "How good is our specification? Was it implemented properly?"

A definition (7) identifying differences between expected and actual results is valuable because it focuses on the fact that when we are testing we need to be able to anticipate what is supposed to happen. It is then possible to determine what actually does happen and compare the two. Without this comparison one of the fundamentals of testing is missing.

The IEEE/ANSI definitions of testing

(1) The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component (IEEE/ANSI, 1990 [Std 610.12-1990]).

(2) The process of analyzing a software item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of the software items (IEEE/ANSI, 1983 [Std 829-1983]).

ANSI uses these two different definitions in the official standards, both of which are somewhat unsatisfactory and which appear to be validation-orientated. The first one, for instance, refers to "operating...under specified conditions", which implies that no testing can be done until there is something to operate.

For the purposes of this book, we want to make sure that our definitions of testing do not preclude the testing of documents like specifications. One kind of testing, verification, can be performed on any work product – any intermediate product in the development cycle. The other kind of testing, validation, can only be performed by executing the code (see Chapter 5).

Managers have their own views and definitions of testing, which reflect different priorities and different pressures from those of the testers. Managers want to know that the product is safe and reliable and that it will function properly under normal and adverse conditions. Managers also want to know that the product is what the users want, so they can ship it and start making some money.

Quality is also important to managers. They want to establish confidence in their products to get future business. For them financial issues will, quite properly, always play an important part in decisions, and they want to be sure they won't get sued.

Most importantly, what about testers themselves? What is a good working definition, that will focus testers' attention on the essentials of the testing job?

The best definition for the tester
The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. (Myers, 1979)

Good testers have a testing attitude

What does testing mean to testers?

Testers hunt errors

If the testers' job is to find every conceivable fault or weakness in the work product, then a good test is one that has a good probability of detecting an as yet undiscovered error, and a successful test is one that detects an as yet undiscovered error.

The focus on showing the presence of errors is the basic attitude of a good tester. It is our job, and it is what gives us personal satisfaction. We feel good on the days when we find defects, and we're thrilled when we've exceeded our error-finding goal for the day or the week. Detected errors are celebrated – for the good of the product.

Testers are destructive – but creatively so

Testing is a positive and creative effort of destruction. It takes imagination, persistence and a strong sense of mission to systematically locate the weaknesses in a complex structure and to demonstrate its failures. This is one reason why it is particularly hard to test our own work. There is a natural real sense in which we don't want to find errors in our own material.

When we test our own material we start with the viewpoint that it works. It works the way we've constructed it. When trying to test our own material we know where the boundaries and the problems are, but we don't think they're big problems, or that anyone will have to confront them (or their consequences) at some later stage. As the developer of the material, our "mind set" is that it's good the way it is, so we test it, but not too hard. What we need is someone else who can attack it with the attitude of: "I'm here to destroy this thing. I'm going to find the defects that I know are there; that's my job and that's what I'm paid to do."

be with the testers who are in a good position to understand their own particular contribution to quality (see Chapter 13).

How testers do it

Having gotten the "tester attitude," how do we go about detecting errors?

- by examining the internal structure and design?
- by examining the functional user interface?
- by examining the design objectives?
- by examining the users' requirements?
- by executing code?

The answer is that we have to do all these things, and many more.

In the real world, under schedule pressure from customers and others, senior management can set deadlines that are frequently unrealistic and therefore strongly counterproductive as far as the final objective of customer satisfaction is concerned. Managers often feel the pressure to produce zero-defect software, even though no one has ever done it. Productivity may be low; an inadequate process may have been inherited which has not been fixed and sufficiently automated, and it is difficult to know where to start. Against this background, testing practitioners and managers may well ask: "What do we do first?"

The answer is to start in the right place for your particular organization. Go for small gains. It isn't necessary to have all of Total Quality Management in place to make solid improvements in the development and testing processes. Before going any further, you should add some items to the potential improvement list (see Chapter 3 and Appendix H). Some possible inclusions derived from this chapter are listed below.

What can we do now?

Here are suggestions for items to include on your "clean sheet" described in Chapter 3.

- Discuss and obtain consensus on your organization's definition of testing. Use the definitions in this chapter as a starting point.
- When you next receive requirements, organize a meeting between the developers and testers to discuss them.
- Work for management support for more team spirit between testers and developers.

Chapter 5

Critical choices: what, when, and how to test

Good testers are always capable of designing more tests than would ever be practical to implement and execute. Exhaustive testing would mean that many products never reach the market because testing would never be completed, and even if they did, they would be astronomically expensive.

Fifty per cent or more of the development organization's time is frequently devoted to error detection and removal, and in most organizations this massive use of resources is neither properly documented nor strategically justified. The real-world solution is to make choices, but on what should they be based?

One choice is to be sure we test the right things – to ensure the most critical items are tested and not to waste limited testing resources on less important items. Another choice is to test early – to focus on detecting the errors closer to the phase where they are introduced, in order to prevent the costly migration of errors downstream.

Testing has a life cycle that parallels the development cycle, one that includes both verification testing and validation testing. Testing processes should, as far as possible, be integrated at the most effective points in the development life cycle.

Testing resources can rarely be dramatically increased in the short term, so we have to use the ones we have in the best possible way.

Risk and risk management

Why would we hesitate before:

- making a parachute jump?
- using an experimental drug?
- investing $20,000 in the stock of a company?
- piloting a brand new aircraft on its initial flight?
- attempting to repair a high-voltage line?
- loaning a friend, or worse yet, a relative $10,000?

- disarming a bomb?
- hiring a teenager as a house sitter?

On the face of it, these undertakings are hazardous. They are risky. We need to work out how risky they are before we do them, and take all reasonable precautions against their coming out badly.

Risk is the probability that undesirable things will happen, such as loss of human life, or large financial losses. The systems we develop, when they don't work properly, have consequences that can vary from the mildly irritating to the catastrophic. Testing these systems should involve informed, conscious risk management.

We can't do everything. We have to make compromises, but we don't want to take risks that are unacceptably high. Key questions must be asked: "Who is going to use the product? What is it being used for? What is the danger of it going wrong? What are the consequences if it does? Is it loss of money? Is it a loss of customer satisfaction? Is it loss of life?"

For each product, we must implement the most cost-effective testing that will ensure that it's reliable enough, safe enough and meets the user/customer requirement. This may sound easy, but examples show that there can be hundreds of test cases for just a few lines of code. If every one of these cases took half a second to execute, it would still take decades to run all the possible test cases for a major product.

This doesn't mean we have to give up testing. It simply means that we never have enough time to test everything completely, so decisions have to be made based on the risks involved. When risk is used as a basis for testing choices, we are doing the rational thing and choosing the parts of the system that have the most serious consequences and focusing our attention on these.

Another basis for choice of testing focus is frequency of use. If a part of the system is used often, and it has an error in it, its frequent use alone makes the chances of a failure emerging much higher.

It is also rational to focus on those parts of the system or program that are most likely to have errors in them. There are many reasons why there may be extra risks involved in a particular piece of software, but the point is to look at them and make informed decisions based on that observation. The reason may be inherent in how that part of the system was created: it may be because the development team is constructed in a certain way and a junior person has written a complex piece of code, so that there's an extra risk in that part of the product; it could have to do with the schedule (too tight); it could have to do with the development resources (insufficient); it could have to do with the budget (too tight again).

In practice, this kind of risk analysis is often a matter of basic communication. If we talk to the development managers and product developers and ask what they are worried about, and their answer is: "Well, we've got an impossible schedule, and we have this very complex part of the system and we're not sure if we've got it right", then we as testers should treat this as an indication that we should give it special attention (see Figure 5.1).

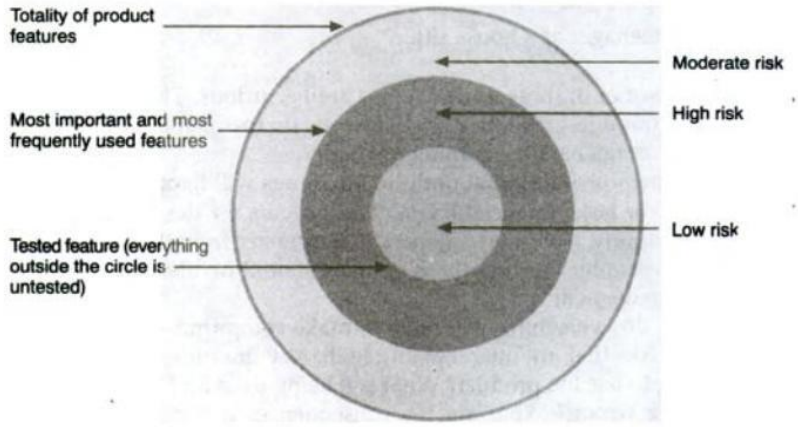

Figure 5.1 Risks in the system as a basis for testing choices. The key questions in this figure involve the "frequently used features." How accurately have they been identified? How close to reality are they? If they are close, then the relative proportions of risk may also be close. If they do not reflect reality, then the "high risk" area could be much larger than that shown. (© 1993, 1994 Software Development Technologies)

For each testing activity to be performed, testing objectives are prioritized by making judgments on the potential for all kinds of failure situations. Of 10 distinct testing activities (four in verification and six in validation – see Chapters 7 and 8), decisions must be made, based on the size, complexity and criticality of the product, about which activities (if any) may be skipped. In any event, we must aim to test to the degree that the risks are unacceptably high.

Risk is not just the basis for making management decisions on testing. It is the basis for decisions that test practitioners make every day. From Figure 5.1 it is clear that this is not a simple matter. But that doesn't mean that any attempt to assess the risk should be abandoned. Rather we should use the parameters spelled out in this section as a starting point for a more realistic assessment of the appropriateness of our present testing effort.

Start testing early

Each development phase (or translation) within the software development process creates a work product that can be tested to see how successful the translation is. At the early stages these work products are requirements and specifications, and they are available to be read and compared with other documents. Later in the process there is code that can be executed. It can,

of course, also be reviewed on paper. If there is a choice, test early, because early is where many of the most important errors are, and this point is reinforced by the fact that more than 50% of all defects are usually introduced in the requirements stage alone.

Test early and prevent defect migration. Every time there is an opportunity to find a defect and we don't find it, and it is allowed to migrate to the next stage, it's going to cost much more to fix when we do find it. At the next stage, it can cost an order of magnitude more and an order of magnitude more again at the stage after that. The cost is maximized if the error is detected after the product is shipped to the customer and minimized if it's detected in the phase where it's introduced.

In the real world of projects under pressure and in a less than ideal organization, what can we do about poor requirements? They can be verified using the review methods in Chapter 7. Inspections can be started on a small scale. They have proven to be winners for more than 20 years in removing a high proportion of the errors. Even before proper verification is implemented, informal methods can help a lot.

Simple, better communication is one element. Customers change their minds about what they want. That is part of the process. But as testers and developers, we can be much more careful about how we record their requirements, how we review them and how we get agreement on them.

We can also facilitate the kind of communication that is so often missing by insisting on getting the right people to meet (product developers, customers, testers, and marketing) to talk about requirements and specifications, and record them meticulously. Often much of this communication will be about language and defining terms: "This is the deliverable, and this is who is responsible for that deliverable, and this is how we change that deliverable, and as it's critical, this is how we are going to test it." At this point, we are starting to touch base with the real software engineering process and maturity issues.

Basic forms of the testing process – verification and validation

Testing can be separated into two basic forms. Given the value of testing early, these definitions of testing are not only confined to testing code, but also include the testing of documents and other non-executable forms of a product.

Verification, as defined by IEEE/ANSI, is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Do we have clear and complete requirements? Is there a good design, and what are the work products produced in the design? Verification is the process of evaluating, reviewing, inspecting, and doing desk checks of work products such as requirement specifications, design specifications, and code. For code it means the static analysis of the code – a code review – not the dynamic execution of the code. Verification testing can be applied to all those things that can be reviewed in the early phases to make sure that what comes out of that phase is what we expected to get.

Some people call verification "human" testing, because it usually involves looking at documents on paper. By contrast, validation usually involves the execution of software on a computer.

> *Validation,* as defined by IEEE/ANSI, is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

Validation normally involves executing the actual software or a simulated mock-up. Validation is a "computer-based testing" process. It usually exposes symptoms of errors.

It is a central thesis of this book that testing includes both verification and validation.

> *Definition*: Testing = verification plus validation

Verification and validation are complementary. The effectiveness of error detection suffers if one or the other is not done. Each of them provides filters that are designed to catch different kinds of problems in the product.

Historically testing has been, and continues to be, largely validation-orientated. It is not that we should stop doing validation, but we want to be a lot smarter about how we do it, and how we do it in combination with verification. We must also ensure that we do each of them at the right time on the right work products.

Testing, the development cycle, and the real world of contracts

Depending on the type of organization and other circumstances, there are variations to the software life-cycle model, but they all have much in common. Figure 5.2 shows a typical model for the development life cycle and the place of testing within it.

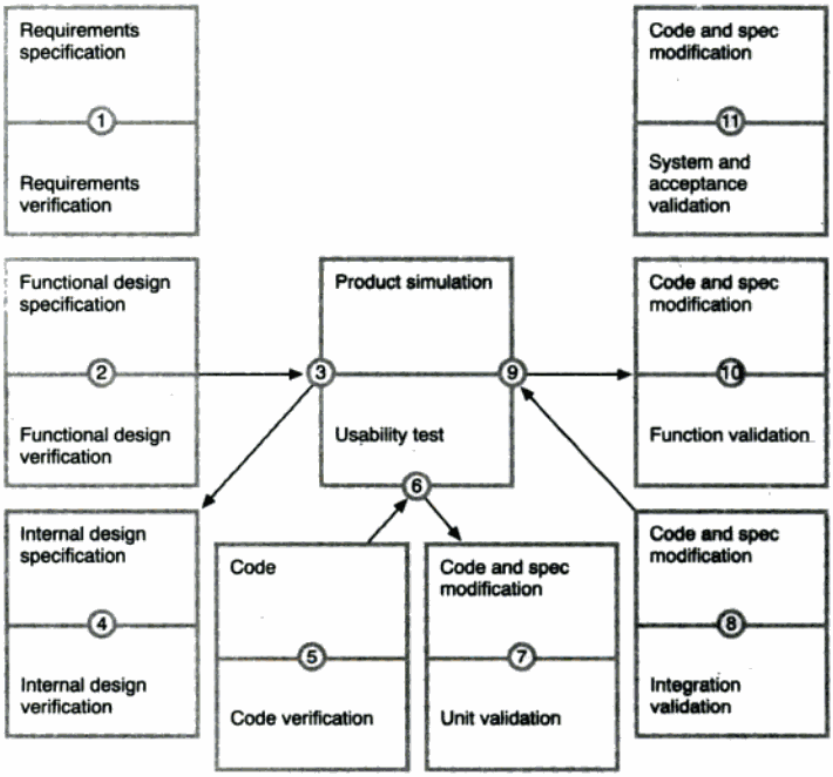

Figure 5.3 The Software Development Technologies (SDT) Dotted-U Model above shows the integration of the development and the test process. This book clearly describes all of the components shown in the model. (© 1993, 1994 Software Development Technologies)

for any testing activity may occur before that activity is depicted in the figure. Each of the 10 fundamental testing activities shown (four in verification and six in validation) is covered in detail.

Effective testing ...

Effective testing removes errors. In any particular case, how do we know how much testing we should aim to do? Should we do full testing or only partial testing?

The basic forms of testing are as follows:

(1) *Full testing* starts no later than the requirements phase, and continues through acceptance testing.
(2) *Partial testing* begins any time after functional design has been completed, with less than optimal influence on requirements and functional design.
(3) *Endgame testing* is highly validation orientated, with no influence on requirements or functional design.
(4) *Audit-level testing* is a bare-bones audit of plans, procedures, and products for adequacy, correctness, and compliance to standards. (Lewis, 1992)

Once again, a big part of the answer lies in risk management. To make effective risk management decisions, we need a clear definition for critical software.

Critical software, as defined by IEEE/ANSI (1990 [Std 610.12-1990]), is software whose failure could have an impact on safety, or could cause large financial or social losses.

Use full testing for critical software or any software that will have heavy and diverse usage by a large user population.

Use partial testing for small, non-critical software products with a small, captive user population (i.e., where the consequences of failures are never damaging).

To enable an effective test effort, we need a software development process that produces:

- *requirements specifications* (required for full testing);
- *functional design specifications* (required for full, partial, and endgame testing);
- *internal design specifications* (required for maximum effectiveness of full and partial testing).

From a testing perspective, full testing means we must have reasonably clear and complete requirements specifications. One of the first things testers can do to improve their process is to press for access to the kind of specification that they need.

For partial or endgame testing we need at least the functional design specifications. For maximum effectiveness we need the internal design specifications because internal information of this kind tells us how the product has been constructed.

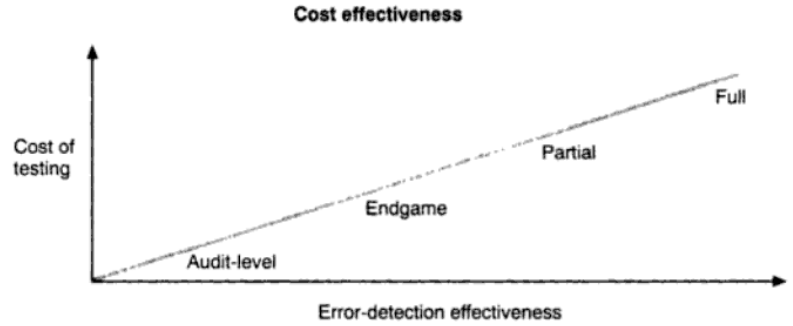

Figure 5.4 The more effective the error detection, the greater the savings in development and maintenance costs over the life of the product. (© 1993, 1994 Software Development Technologies)

… and cost-effective testing

Effective testing is testing that is successful at detecting errors. But this may not be an exercise that is always worthwhile from a financial point of view. We have considered the cost of errors (what they cost if they remain in the product), but in a real-world environment we need to confront and make explicit a number of questions about the cost effectiveness of our testing effort.

Do we know what testing is really costing us? How is it affecting development time? Do we know what percentage of our development resource testing represents? Are testing tools saving us time? Are they paying for themselves? Are we trained to exploit them to the full, or have they already become shelfware? Are we using our testing resources to find errors that represent the biggest financial risk? If we look at the overall impact on our business objectives of testing the products of our development, is our testing cost effective? (See Figure 5.4.)

When we start to ask these kinds of questions within real-world organizations, we find the gulf between awareness and action to be very wide indeed. In other words, many people are aware they ought to know the answers to these questions, but they know that they are far away from doing so, usually because they are too busy fighting today's fires.

Frequently, they believe that these kinds of questions are management's problem, as indeed they are. In practice, questions of the cost effectiveness of the testing process rarely get considered. (For more detail on cost effectiveness and its measurement, see Chapter 12.)

Chapter 6

Critical disciplines: frameworks for testing

Any software engineering method that is common practice in software development projects is usually also applicable to software testing efforts. This means that the disciplines of planning and analysis, formal documents, configuration management, measurement, standards, tools, and procedures can be applied to testing.

As projects become more complex, and development and testing groups become larger, these disciplines become increasingly critical to success.

Most of these ideas come together in the notion of *process maturity*. The more of these disciplines we are doing, and the more we are doing systematically, the more mature our software process. Most of them are handled in greater detail later in the book.

Planning

Without a plan, we don't know:

- what we're going to do
- how we're going to do it
- how long it will take
- the resources required
- the cost.

It is easy to see why medium-sized (let alone large) software development projects become so disastrously out of control if the planning is not done properly. Planning is the first step of every testing activity.

If testing is to be included as a serious part of software engineering, testers have to take time to analyze the product. Product requirements must be reviewed before tests are designed. Tests should be designed before implementation is started. In fact, the process of designing tests helps to detect specification defects. Specifications are therefore improved by a good

testing process. In other words, we have to plan, and once we start planning, all kinds of other questions come up.

What kinds of standards are going to be useful? Are we going to do to inspections and reviews? What kinds of tools should we be using to find the most critical problems in this particular product? Do we have the time and financial resources to cover the testing we have to do? What kind of review are we going to have of our processes as testers? Do we have our priorities right?

There is a general human tendency to want to "get on with the next thing," especially under time pressure. For testers this usually means working the code and finding errors in it. However, resources spent intelligently on the earlier testing phases are repaid many times over. Time spent on early planning is never wasted, and usually the total time cycle is significantly shorter. There are two distinct stages, planning and execution, of each activity within verification. This is the case regardless of whether we are doing requirements verification, functional design verification, internal design verification, or code verification.

The considerations in verification planning

For each type of verification (requirements, functional design, internal design, code) the issues to be addressed are as follows:

- the verification activity to be performed;
- the methods used (inspection, walkthrough, etc.);
- the specific areas of the work product that will and will not be verified;
- the risks associated with any areas that will not be verified;
- prioritizing the areas of the work product to be verified;
- resources, schedule, facilities, tools, and responsibilities.

The considerations in validation planning

Validation testing activities include unit testing (by development), integration testing (by development), usability testing, function testing, system testing, and acceptance testing. The tasks associated with this are high-level planning for all validation activities as a whole and testware architectural design.

For each validation activity we have to do:

- detailed planning
- testware design and development
- test execution
- test evaluation
- testware maintenance.

For validation planning the issues to be addressed are as follows:

- test methods
- facilities (for testware development vs. test execution)
- test automation
- testing tools
- support software (shared by development and test)
- configuration management
- risks (budget, resources, schedule, training).

(There is more detail on planning and master test plans in Chapter 10.)

Software engineering maturity and the SEI

The Software Engineering Institute (SEI) is a federally-funded research and development center, sponsored by the Department of Defense, and affiliated with Carnegie Mellon University. It was established by Congress in 1984 to address a twofold shortage of trained software professionals and quality software, produced on schedule and within budget.

The mission of the SEI is to provide leadership to advance the state-of-the-practice of software engineering to improve the quality of systems that depend on software. The SEI focuses on software process, on the assumption that improving the process, which means bringing engineering discipline to the development and maintenance of software, is the key to better quality software products. This is consistent with the basic premise of this book – the importance of the software testing process.

SEI process maturity levels

To assess the ability of development organizations to develop software in accordance with modern software engineering methods, the SEI defines five process maturity levels as part of a process model called the Capability Maturity Model (CMM):

Level 1: *Initial*	(Anarchy)	Unpredictable and poorly controlled
Level 2: *Repeatable*	(Folklore)	Can repeat previously mastered tasks
Level 3: *Defined*	(Standards)	Process characterized, fairly well understood
Level 4: *Managed*	(Measurement)	Process measured and controlled
Level 5: *Optimizing*	(Optimization)	Focus on process improvement

Few organizations are beyond level 1 on the maturity scale, which is characterized as consisting of unpredictable, poorly-controlled processes. To achieve level 2, previously mastered tasks have to be repeatable. In other words, we have the disciplines in place for proper project control, project planning, reviews, and communication control.

While general software process improvement and maturity are beyond the scope of what many testers do every day, the SEI CMM describes goals and activities that are of vital importance to anyone striving to improve their everyday testing process. In fact, activities associated with level 3 that specifically relate to testing include the following:

Level 3, Activity 5: Software testing is performed according to the project's defined software process

(1) Testing criteria are developed and reviewed with the customer and the end user, as appropriate.
(2) Effective methods are used to test the software.
(3) The adequacy of testing is determined by:
    (i) the level of testing performed (e.g., unit testing, integration testing, system testing),
    (ii) the test strategy selected (e.g., black-box, white-box), and
    (iii) the test coverage to be achieved (e.g., statement coverage, branch coverage).
(4) For each level of software testing, test readiness criteria are established and used.
(5) Regression testing is performed, as appropriate, at each test level whenever the software being tested or its environment changes.
(6) The test plans, test procedures, and test cases undergo peer review before they are considered ready for use.
(7) The test plans, test procedures, and test cases are managed and controlled (e.g., placed under configuration management).
(8) Test plans, test procedures, and test cases are appropriately changed whenever the allocated requirements, software requirements, software design, or code being tested changes.

Level 3, Activity 6: Integration testing of the software is planned and performed according to the project's defined software process

(1) The plans for integration testing are documented and based on the software development plan.
(2) The integration test cases and test procedures are reviewed with the individuals responsible for the software requirements, software design, and system and acceptance testing.

(3) Integration testing of the software is performed against the designated version of the software requirements document and the software design document.

Level 3, Activity 7: System and acceptance testing of the software are planned and performed to demonstrate that the software satisfies its requirements

(1) Resources for testing the software are assigned early enough to provide for adequate test preparation.
(2) System and acceptance testing are documented in a test plan, which is reviewed with, and approved by, the customer and end users, as appropriate.
(3) The test cases and test procedures are planned and prepared by a test group that is independent of the software developers.
(4) The test cases are documented and are reviewed with, and approved by, the customer and end users, as appropriate, before the testing begins.
(5) Testing of the software is perfomed against baselined software and the baselined documentation of the allocated requirements and the software requirements.
(6) Problems identified during testing are documented and tracked to closure.
(7) Test results are documented and used as the basis for determining whether the software satisfies its requirements.
(8) The test results are managed and controlled.

How process maturity affects testing

There are strong indications that the amount of testing budget spent on non-technical (management) issues is in inverse proportion to the maturity level within the environment as a whole (see Figure 6.1).

Level 1 organizations are characterized by so much uncertainty that the outcome is unpredictable. To the extent that development doesn't know where it is going, testing cannot plan its own activities and will spend more time working on non-technical issues.

There are special consultancy service organizations trained in the SEI model which conduct software process assessments (SPAs) on request. SPA is a method developed by the SEI to help organizations evaluate their software process maturity and identify key areas for improvement. They look at the practices an organization is using and recommend practices that should be implemented now to improve processes.

SPA is based on a series of questions specific to particular subject areas, and a number of follow-up questions that expand on the information obtained. Answers are weighted according to the importance of their subject matter (see Humphrey, 1989).

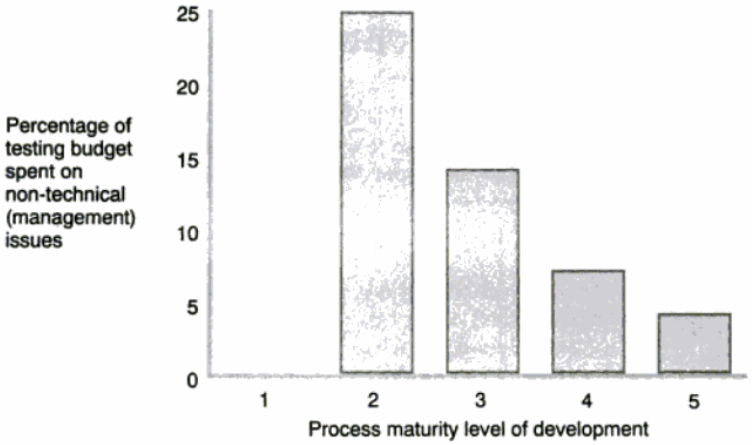

Figure 6.1 How the process maturity level affects testing (Lewis, 1992). To the extent that development doesn't know where it is going, testing cannot plan its own activities and will spend more time on management issues rather than technical issues. (© 1993, 1994 Software Development Technologies)

More detail on SEI assessment

SEI evaluation questions are divided into seven sections. The following sample questions all apply to organizations at Level 1 which are aiming for Level 2.

Organization:
Is there a software configuration control function for each project that involves software development?

Resources, personnel, and training:
Is there is a required training program for all newly appointed development managers designed to familiarize them with software project management?

Technology management:
Is a mechanism used for maintaining awareness of the state of the art in software engineering technologies?

Documented standards and procedures:
Is a formal procedure used in the management review of each software development prior to making contractual commitments?

Process metrics:
Are software staffing profiles maintained of actual staffing versus planned staffing?

Data management and analysis:
Is review efficiency analyzed for each project? (level 4 question)

Process control:
Is a mechanism used for controlling changes to the software requirements?

Configuration management

Configuration management (CM) problems stem from confusion and lack of control. These problems can waste enormous amounts of time; they often happen at the worst possible moment; they are almost always very frustrating, and they are totally unnecessary.

If...
... we can't identify the source code that corresponds to a given module of object code;
... we can't determine which version of the COBOL compiler generated a given object module;
... a bug that was corrected last month suddenly reappears;
... we can't identify the source-code changes that were made in a particular revision of software;
... simultaneous changes are made to the same source module by multiple developers, and some of the changes are lost;
... shared (source) code is changed, and all programmers sharing the code are not notified;
... changes are made to the interface of common (runtime) code, and all users are not notified;
... changes are made, and the changes are not propagated to all affected versions (releases) of the code...
... then what is lacking is configuration management.

What is CM?

Configuration management (CM) is a four-part discipline applying technical and administrative direction, control, and surveillance for:

(1) *Configuration identification*
  (i) conventions for identifying baseline and revision levels for all program files (source, object listing) and revision-specific documents;
  (ii) derivation records identify "build" participants (including source and object files, tools and revision levels used, data files).
(2) *Configuration and change control*
  (i) safeguard defined and approved baselines;
  (ii) prevent unnecessary or marginal changes;
  (iii) expedite worthwhile changes.
(3) *Configuration status accounting*
  (i) recording and tracking problem reports, change requests, change orders, etc.

that testers and others doing verification have a framework for what they can expect to find. More specifically, standards tell us what to put into key test documents, such as a test plan.

Standards are not only practical from the development point of view, but they are increasingly the basis for contracts and therefore also, when things go wrong, for litigation. One of the issues that arises in litigation is whether the software was developed according to known standards that are prevalent in the industry today. This means we need to know not only what the standards are, but to also see that they are applied.

IEEE/ANSI standards

Many key standards relating to software testing are generated by the Institute of Electrical and Electronics Engineers (IEEE). A transnational organization founded in 1884, IEEE consists of dozens of specialized societies within geographical regions throughout the world. Software testing standards are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board.

These standards are created through a process of obtaining the consensus of practicing professionals. This consensus process, which includes careful discussion and debate among members of the various committees who serve voluntarily, is one of the fundamental themes of the standards process. Another key theme is to provide standards in a timely manner; from project approval to standard approval is approximately three years.

Key US software testing standards

- IEEE Standard for Software Test Documentation, Reaff. 1991 (IEEE/ANSI Std 829-1983)
- IEEE Standard for Software Unit Testing, Reaff. 1993 (IEEE/ANSI Std 1008-1987)

Other standards related to software testing

- IEEE Standard for Software Verification and Validation Plans, Reaff. 1992 (IEEE/ANSI Std 1012-1986)
- IEEE Standard for Software Reviews and Audits (IEEE/ANSI Std 1028-1988)
- IEEE Standard for Software Quality Assurance Plans (IEEE/ANSI Std 730-1989)
- IEEE Standard Glossary of Software Engineering Terminology (IEEE/ANSI Std 610.12-1990)

One of the earliest and perhaps still the most important software testing standard, the Standard for Software Test Documentation was initially

approved by IEEE in 1982 and by ANSI in 1983, and was reaffirmed in 1991. This standard describes the content and format of basic software testing documents. It is often the first standard to be used by test practitioners as the basis and template for key test documents such as test plans, test design specifications, and test summary reports.

The other key standard, the Standard for Software Unit Testing, specifies a systematic approach to software unit testing and provides guidance to assist with the implementation and usage of unit testing. The standard describes a testing process composed of a hierarchy of phases, activities, and tasks. It defines a minimum set of tasks for each activity.

IEEE uses one of three designations in every document title:

(1) *Standard* means "must use";
(2) *Recommended practice* means "should use";
(3) *Guide* means "use at your discretion".

These standards provide recommendations reflecting the state of the practice in software testing. Think of it as receiving extremely cost-effective advice from concerned professionals who have thought long and hard about key issues facing those both new and old to software engineering.

Once approved by IEEE, many of these standards go on to be approved as American National Standards Institute (ANSI) standards. It is the coordinator of America's voluntary standards system. ANSI will approve standards as American National Standards after it has verified evidence that consensus exists. ANSI verifies that all substantially affected interests have been given an opportunity to participate in the development of a standard or to comment on its provisions and that their views have been carefully considered.

More detail on US standards

The IEEE/ANSI software engineering standards can be purchased through IEEE.

Contact for IEEE:
IEEE
445 Hoes Lane
PO Box 1331
Piscataway, NJ 08855-1331 USA
Freephone: (1) 800 678 IEEE
From outside USA: (1) 908 981 0060

(3) *Internal design specification.* Insist that development produce an internal design specification. Some organizations don't.

A mature development organization will produce these documents and faithfully maintain them for the life of the product. With proper configuration management, any change is reflected in all affected documents. These documents are under change control and kept in lockstep with the code.

Testing produces its own formal documents corresponding to its own life cycle, and these form part of testware.

Testware

A deliverable is a major work product of an organization. Hardware development engineers produce hardware. Software development engineers produce software. Software test engineers produce testware.

Testware is produced by both verification and validation testing methods. Testware includes verification checklists, verification error statistics, test data, and supporting documentation like test plans, test specifications, test procedures, test cases, test data, and test reports.

Testware is so named because it has a life beyond its initial use. Like software, it should be placed under the control of a configuration management system, saved, and faithfully maintained. Contrary to popular perception, testing is not an on-the-fly activity with no tangible remains when the work is complete. Like software, testware has significant value because it can be reused without incurring the cost of redevelopment with each use.

It is important to maintain testware, just as software and hardware are maintained. Part of the tester's job is to create testware that is going to have a specified lifetime and is a valuable asset to the company, controlled and managed like any other. When testware is being created, it has to be put under some kind of control so that it is not lost. If a particular tester leaves the company, someone else can then maintain and continue the process.

Giving testware a name helps to give validity to a test organization that doesn't usually think in terms of having a specific deliverable. It gives ownership to what testers do. Professional testers have a product. Testers have test libraries, their own code and their own software. Testers' specific deliverables are testware. The concept of testware can make it easier for testers to communicate with each other and with other organizations.

Testware is referred to throughout this book. Testware management, though not directly related to error detection, has an enormous impact on the efficiency of the test effort. There is more detail on verification testware in Chapter 7, and on testing deliverables at each phase of the life cycle and the management of testware in Chapter 9.

Measurement

Without measurements, we never know whether we're moving forward or backward or standing still. It is no coincidence that a major section of the SEI Software Process Assessment questionnaire is devoted to process metrics.

Sample process metrics questions

- Are statistics of software errors gathered?
- Are profiles maintained of actual versus planned test cases for groups completing testing?
- Are software trouble reports resulting from testing tracked to closure?
- Are design and code review coverage measured and recorded?

Mature development and testing organizations are able to provide metrics on all key elements in their process. This is not only because they need to monitor their performance, their use of resources and their efficiency as a basis for management decisions, but also because even asking the questions relating to collecting the statistics is a major springboard for process improvement, at any level.

While measurement can easily look like more unnecessary bureaucracy, there are a number of big questions for testing to which the development of measures within the organization is the only way to get answers, and thus make good decisions in the future.

Measurement can tell us ...

- What was the (real) size of the test effort on a given product?
- How efficient were our verification efforts?
- How thorough were our validation tests?
- What was the quality of the product:
  - during test (before customer use)?
  - in production use by customers?
  - compared to other products?
- How many (approximately) errors are there in the product:
  - before testing begins?
  - later, after some experience with the product?
- When do we stop testing?

Measurement provides an idea of the real program complexity. It helps in planning the test effort, and in predicting how many errors there are and

where they occur. Measurement of the number and types of errors detected provides an accurate idea of our efficiency at verification and the weaknesses in our development process.

Measuring validation test coverage provides quantitative assessments of the thoroughness and comprehensiveness of our validation tests. Tracking test execution status monitors the convergence of test categories and provides quantitative information on when to stop testing.

Measuring and tracking incident reports (by severity category):

- provides a leading indicator of product quality;
- provides significant criteria for release readiness;
- correlates to users' satisfaction with the product;
- serves as a predictor of the number of remaining errors;
- when normalized, provides a measure of product quality relative to other products;
- provides a measure of testing efficiency (customer reported vs. testing reported incidents).

Remember that

"You can't control what you can't measure."

Tom DeMarco, 1982

(See Chapter 12 for more details on useful measures and how to implement them.)

Tools

Testing tools, like any other kinds of tools, provide leverage. There is a wide variety of tools available today to provide assistance in every phase of the testing process. If we are to maximize the benefit we get from tools, there are a number of questions to be asked as part of implementing an effective tools program:

(1) How do the tools fit into and support our test process?
(2) Do we know how to plan and design tests? (Tools do not eliminate the need to think, to plan, and to design.)
(3) Who will be responsible for making sure we get the proper training on our new tool?
(4) Who will promote and support tool use within the organization on an ongoing basis?

In other words, like all testing activities, the use of tools needs to be integrated with the larger picture of our development process.